

## 2. Getting Started

Created: April 1, 2003  
Updated: February 4, 2004

### Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

### Introduction

This section is intended as a bird's-eye view of the Toolkit for new users, and to give quick access to important reference links for experienced users. It lays out the general roadmap of tasks required to get going, giving links to take the reader to detailed discussions and supplying a number of simple, concrete test applications.

**Note:** Much of this material is platform-neutral, although the discussion is Unix-centric. Windows or Mac users would also benefit from reading the installation instructions specific to those systems and, where applicable, how to use CVS with MS Windows and Mac OS.

### Chapter Outline

The following is an outline of the topics presented in this chapter:

- Quick Start
- Example Applications
- Source Tree Availability
  - FTP Availability
  - CVS Availability
  - Availability via Shell Scripts
- Source Tree Contents
  - Top-Level Source Organization
  - The Core NCBI C++ Toolkit

- Source Tree for Individual Projects
- The Makefile Templates
- The New Module Stubs
- Decide Where You Will Work (in-tree, in a subtree, out-of-tree)
- Basic Installation and Configuration Considerations
- Basics of Using the C++ Toolkit
  - Compiling and Linking with make
  - Makefile Customization
  - Basic Toolkit Coding Infrastructure
  - Key Classes
  - The Object Manager and datatool
  - Debugging and Diagnostic Aids
  - Coding Standards and Guidelines
- Noteworthy Files

## Quick Start

---

A good deal of the complication and tedium of getting started has thankfully been wrapped by a number of shell scripts. They facilitate a 'quick start' whether starting anew or within an existing Toolkit work environment. ('Non-quick starts' sometimes cannot be avoided, but they are considered elsewhere.)

### 1. **Get the Source Tree (see Figure 1)**

- Download via FTP, **or**
- Run ***cvs\_core.sh*** (requires a CVS repository containing the C++ Toolkit; for NCBI users, the repository is at `$CVSROOT/internal/c++`)

### 2. **Configure the build tree (see Figure 2)**

- Use the ***configure*** script, **or**
  - Use a compiler-specific wrapper script ***scripts/compilers/\*.sh***.
3. **Build the C++ Toolkit from *makefiles* and *meta-makefiles***(if required)
- `make all_r` for a recursive make, **or**
  - `make all` to make only targets for the current directory.
4. **Work on your new or existing application or library** the scripts ***new\_project.sh*** and (for an existing Toolkit project) ***import\_project.sh*** help to set up the appropriate *make-files* and/or source.

# NCBI C++ source tree

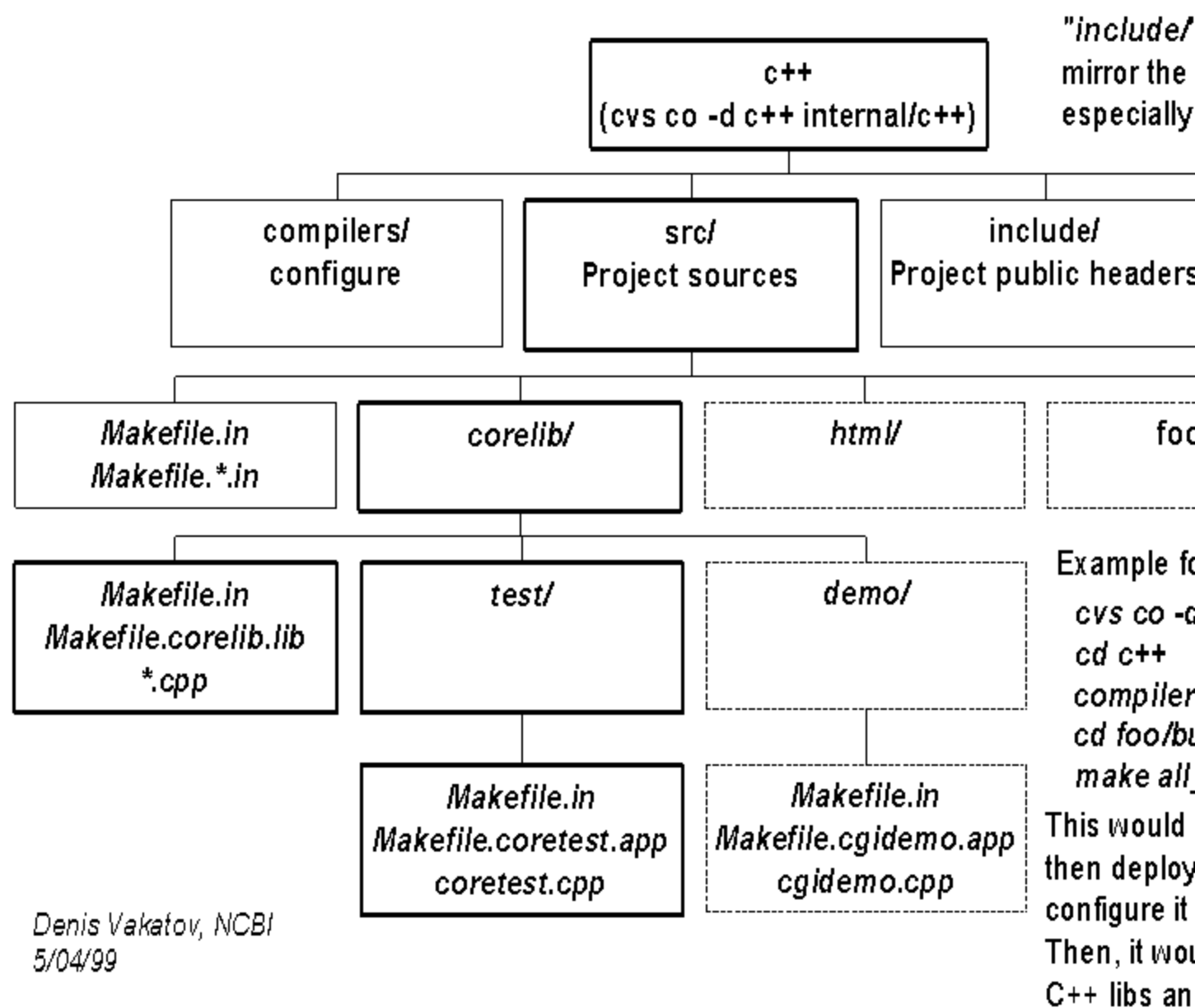


Figure 1: NCBI C++ Source Tree

## NCBI C++ build tree

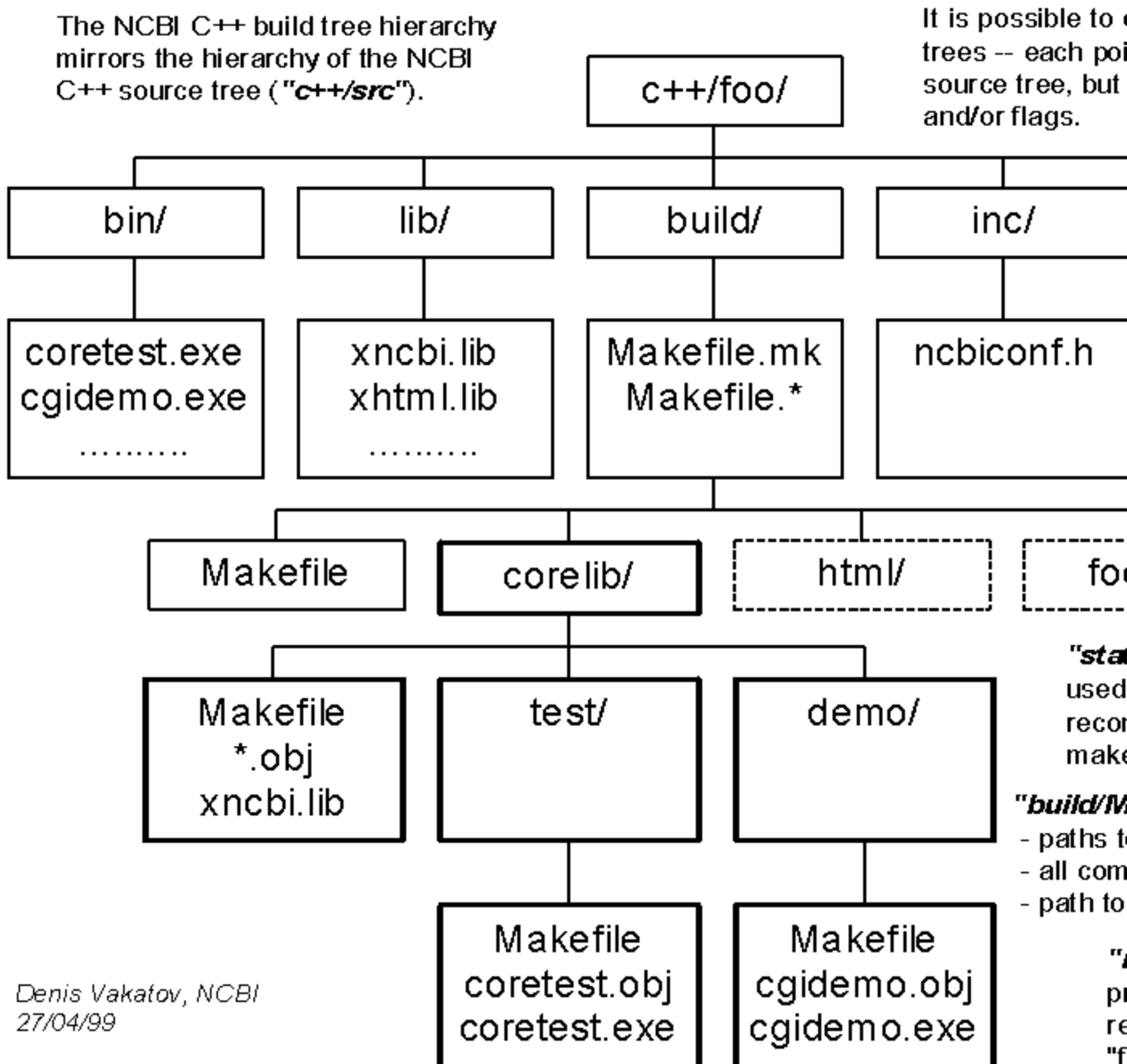


Figure 2: NCBI C++ Build Tree

In a nutshell, that's all it takes to get up and running. The download, configuration, installation and build actions are shown for two cases in this sample.

The last item, employing the Toolkit in a project, completely glosses over the substantial issue of how to use the installed Toolkit. Where does one begin to look to identify the functionality to solve your particular problem, or indeed, to write the simplest of programs? "Basics of Using

the C++ Toolkit" will deal with those issues. Investigate these and other topics with the set of sample applications. See **Examples** or Tests for further cases that employ specific features of the NCBI C++ Toolkit.

## Example Applications

---

The suite of application examples below highlight important areas of the Toolkit and can be used as a starting point for your own development. Note that you may generate the sample application code by running the *new\_project.sh* script for that application. The following examples are now available:

- *app/basic* A generic application to demonstrate the use of key Toolkit classes.
- *app/cgi* Presents the Toolkit infrastructure required to write Web-enabled CGI applications.
- *app/dbapi* Presents the Toolkit infrastructure required to write database applications.
- *app/gui* Presents the Toolkit infrastructure required to write GUI applications.
- *app/objects* Presents the Toolkit infrastructure required to write applications using ASN.1 objects.
- *app/objmgr* The Toolkit manipulates biological data objects in the context of an *Object Manager* class (**CObjectManager**). This example shows how to build an application using the object manager.
- *app/alnmgr* Presents the Toolkit infrastructure required to write alignment manager applications.

To build an example use its accompanying *Makefile*.

## Source Tree Availability

---

The source tree is available through FTP, CVS and by running special scripts. The following subsections discuss these topics in more detail:

- FTP Availability
- CVS Availability
- Availability via Shell Scripts

### FTP Availability

The Toolkit source is available via ftp at [ftp://ftp.ncbi.nih.gov/toolbox/ncbi\\_tools++/CURRENT/](ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools++/CURRENT/) [[ftp://ftp.ncbi.nih.gov/toolbox/ncbi\\_tools++/CURRENT/](ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools++/CURRENT/)], and the archives available, with unpacking instructions, are listed on the download page. If you plan to modify the Toolkit source in any

way with the ftp'd code, it is strongly advised that it be placed under a source code control system (preferably CVS) so that you can rollback to an earlier revision without having to ftp the entire archive once again.

## CVS Availability

NCBI users can obtain the source tree directly from the internal CVS repository. Although CVS is historically a Unix-centric utility, MS Windows and Mac OS users can also work with the CVS tree.

## Availability via Shell Scripts

The various shell scripts in `$NCBI/c++/scripts` tailor the working codebase and can prepare the work environment for new projects. Except where noted, an active Toolkit CVS repository is required, and obviously in all cases a version of the Toolkit must be accessible. Be certain the `$CVSROOT` environment variable points at the correct CVS repository before running the scripts.

- **`cvs_core.sh`** (or **`cvs_core.bat`** under MS Windows). Details on **`cvs_core.sh`** are discussed in a later chapter.
- **`import_project.sh`**. Details on **`import_project.sh`** are discussed in a later chapter.
- **`new_project.sh`. No CVS needed.** Details on **`new_project.sh`** are discussed in a later chapter.
- **`update_core.sh`** and **`update_projects.sh`**. Details on **`update_core.sh`** and **`update_projects.sh`** are covered in later chapter.

## Source Tree Contents

---

The following topics are discussed in this section:

- Top-Level Source Organization
- The Core NCBI C++ Toolkit
- Source Tree for Individual Projects
- The Makefile Templates
- The New Module Stubs

## Top-Level Source Organization

The NCBI C++ Toolkit *source tree* (see Figure 1) is organized as follows:

- *src/* -- a hierarchical directory tree of NCBI C++ projects. Contained within *src* are all source files (*\*.cpp*, *\*.c*), along with private header files (*\*.hpp*, *\*.h*), makefiles (*Makefile.\**, including *Makefile.mk*), scripts (*\*.sh*), and occasionally some project-specific data
- *include/* -- a hierarchical directory tree whose structure mirrors the *src* directory tree. It contains only public header files (*\*.hpp*, *\*.h*). **Example:** *include/corelib/* contains public headers for the sources located in *src/corelib/*
- *scripts/* -- auxiliary scripts, including those to help manage interactions with the NCBI CVS code repository, such as verify links ***import\_project.sh***, ***new\_project.sh***, ***cvs\_core.sh***, etc.
- files for platform-specific configuration and installation:
  - *compilers/* -- directory containing compiler-specific **configure** wrappers (*\*.sh*) and pre-built configuration headers (*ncbiconf.\**) for some platforms
  - **configure** -- a multi-platform configuration shell script (generated from template *configure.ac* using autoconf)
  - various scripts and template files used by **configure**, autoconf and autoheader
- **doc/** -- NCBI C++ documentation, including a library reference, configuration and installation instructions, example code and guidelines for **everybody** writing code for the NCBI C++ Toolkit.

## The Core NCBI C++ Toolkit

The 'core' libraries of the Toolkit provide users with a highly portable set of functionality. The following projects comprise the portable core of the Toolkit:

```
corelib connect cgi html util hello
```

Consult the library reference (Part 3 of this book) for further details.

## Source Tree for Individual Projects

For the overall NCBI C++ source tree structure see Top-Level Source Organization above.

An individual project contains the set of source code and/or scripts that are required to build a Toolkit library or executable. In the NCBI source tree, projects are identified as sub-trees of the *src* and *include* directories of the main C++ tree root. For example, *corelib* and *objects/objmgr* are both projects. However, note that a project's code exists in two sibling directories: the public headers in *include/* and the source code, private headers and *makefiles* in *src*.

The contents of each project's *source tree* are:

- *\*.cpp, \*.hpp* -- project's source files and private headers
- *Makefile.in* -- a *meta-makefile* to specify which local projects (described in *Makefile.\*.in*) and sub-projects (located in the project subdirectories) must be built
- *Makefile.\*.lib, Makefile.\*.app* -- *customized makefiles* to build a library or an application
- *Makefile.\** -- "free style" makefiles
- sub-project directories (if any)

## The Makefile Templates

Each project is built by customizing a set of generic *makefiles*. These generic *makefile* templates (*Makefile.\*.in*) are found in *src* and help to control the assembly of the entire Toolkit via recursive builds of the individual projects. (The usage of these *makefiles* and other configurations issues are summarized below and detailed on the Working with Makefiles page.)

- *Makefile.in* -- makefile to perform a recursive build in all project subdirectories
- *Makefile.meta.in* -- included by all makefiles that provide both local and recursive builds
- *Makefile.mk.in* -- included by all makefiles; sets a lot of configuration variables
- *Makefile.lib.in* -- included by all makefiles that perform a "standard" library build, when building only static libraries.
- *Makefile.dll.in* -- included by all makefiles that perform a "standard" library build, when building only shared libraries.
- *Makefile.both.in* -- included by all makefiles that perform a "standard" library build, when building both static and shared libraries.
- *Makefile.lib.tmpl.in* -- serves as a template for the project *customized makefiles* (*Makefile.\*.lib[.in]*) that perform a "standard" library build
- *Makefile.app.in* -- included by all makefiles that perform a "standard" application build
- *Makefile.lib.tmpl.in* -- serves as a template for the project *customized makefiles* (*Makefile.\*.app[.in]*) that perform a "standard" application build
- *Makefile.rules.in, Makefile.rules\_with\_autodep.in* -- instructions for building object files; included by most other makefiles

## The New Module Stubs

A Toolkit module typically consists of a header (*\*.hpp*) and a source (*\*.cpp*) file. Use the stubs provided, which include boilerplate such as the NCBI disclaimer and CVS revision information, to easily start a new module. You may also consider using the sample code described above for your new module.

## Decide Where You Will Work (in-tree, in a subtree, out-of-tree)

---

Depending on how you plan to interact with the NCBI C++ Toolkit source tree, the Toolkit has mechanisms to streamline how you create and manage projects. The simplest case is to work *out-of-tree* in a private directory. This means that you are writing new code that needs only to link with pre-built Toolkit libraries. If your project requires the source for a limited set of Toolkit projects it is often sufficient to work *in a subtree* of the Toolkit source distribution.

Most users will find it preferable and fully sufficient to work in a subtree or a private directory. Certain situations and users (particularly Toolkit developers) do require access to the full Toolkit source tree; in such instances one must work *in-tree*.

## Basic Installation and Configuration Considerations

---

**Note:** Much of this discussion is Unix-centric, so Windows and Mac users would also benefit from reading the installation instructions specific to those systems.

The configuration and installation process is automated with the **configure** script and its wrappers in the *compilers* directory. These scripts handle the compiler- and platform-dependent Toolkit settings and create the *build tree* (see Figure 2) skeleton. The configured *build tree*, located in *<builddir>*, is populated with customized *meta-makefile*, headers and source files. Most system-dependence has been isolated in the *<builddir>/inc/ncbiconf.h* header. By running **make all\_r** from *<builddir>*, the full Toolbox is built for the target platform and compiler combination.

Summarized below are some basic ways to control the installation and configuration process. More comprehensive documentation can be found at [config.html](#).

- **A Simple Example Build**

- **configure Options** View the list of options by running

```
configure --help
```

- **Enable/Disable Debugging**

- **Building Shared and/or Static Libraries** Shared libraries (DLL's) can be used in Toolkit executables and libraries for a number of tested configurations. Note that to link with the shared libraries at run time a valid *runpath* must be specified.

- **Influencing *configure* via Environment Variables** Several environment variables control the tools and flags employed by *configure*. The generic ones are: `CC`, `CXX`, `cpp`, `AR`, `RANLIB`, `STRIP`, `CFLAGS`, `CXXFLAGS`, `cppFLAGS`, `LDFLAGS`, `LIBS`. In addition, you may manually set various localization environment variables.
- **Multi-Thread Safe Compilation**
- **Controlling Builds of Optional Projects** You may selectively build or not build one of the optional projects ("serial", "ctools", "gui", "objects", "internal") with *configure* flags. If an optional project is not configured into your distribution, it can be added later using the *import\_projects* script.
- **Adjust the Configuration of an Existing Build** If you need to update or change the configuration of an existing build, use the *reconfigure.sh* or *relocate.sh* script.
- **Working with Multiple *build trees*** Managing builds for a variety of platforms and/or compiler environments is straightforward. The configure/install/build cycle has been designed to support the concurrent development of multiple builds from the same source files. This is accomplished by having independent *build trees* that exist as sibling directories. Each build is configured according to its own set of configuration options and thus produces distinct libraries and executables. All builds are nonetheless constructed from the same source code in `$NCBI/c++/{src, include}`.

## Basics of Using the C++ Toolkit

---

The following topics are discussed in this section:

- Compiling and Linking with *make*
- Makefile Customization
- Basic Toolkit Coding Infrastructure
- Key Classes
- The Object Manager and *datatool*
- Debugging and Diagnostic Aids
- Coding Standards and Guidelines

### Compiling and Linking with *make*

The NCBI C++ Toolkit uses the standard Unix utility *make* to build libraries and executable code, using instructions found in *makefiles*. More details on compiling and linking with *make* can be found in a later chapter.

To initiate compilation and linking, run *make*:

```
make -f <Makefile_Name> [<target_name>]
```

When run from the top of the build tree, this command can make the entire tree (with target *all\_r*). If given within a specific project subdirectory it can be made to target just the that project. The Toolkit has in its *src* directory templates (e.g., *Makefile.\*.in*) for *makefiles* and *meta-makefiles* that define common file locations, compiler options, environment settings, and standard make targets. Each Toolkit project has a specialized *meta-makefile* in its *src* directory. The relevant *meta-makefile* templates for a project, e.g., *Makefile.in*, are customized by **configure** and placed in its *build tree*. For new projects, whether in or out of the C++ Toolkit tree, the programmer must provide either *makefiles* or *meta-makefiles*.

## Makefile Customization

Fortunately, for the common situations where a script was used to set up your source, or if you are working in the C++ Toolkit source tree, you will usually have correctly customized *makefiles* in each project directory of the build tree. For other cases, particularly when using the **new\_project.sh** script, some measure of user customization may be needed. The more frequent customizations involve (see "Working with Makefiles" or "Project makefiles" for a full discussion):

- *meta-makefile* macros: APP\_PROJ, LIB\_PROJ, SUB\_PROJ, USR\_PROJ Lists of applications, libraries, sub-projects, and user projects, respectively, to make.
- Library and Application macros: APP, LIB, LIBS, OBJ, SRC List the application name to build, Toolkit library(ies) to make or include, non-Toolkit library(ies) to link, object files to make, and source to use, respectively.
- Compiler Flag Macros: CFLAGS, cppFLAGS, CXXFLAGS, LDFLAGS Include or override C compiler, C/C++ preprocessor, C++ compiler, and linker flags, respectively. Many more localization macros are also available for use.
- Altering the Active Version of the Toolkit You can change the active version of NCBI C++ toolkit by manually setting the variable \$(builddir) in *Makefile.foo\_[app|lib]* to the desired toolkit path, e.g.: `builddir = $(NCBI)/c++/GCC-Release/build`. Consult this list or, better, look at the output of `'ls -d $NCBI/c++/*/build'` to see those pre-built Toolkit builds available on your system.

## Basic Toolkit Coding Infrastructure

Summarized below are some features of the global Toolkit infrastructure that users may commonly employ or encounter.

- The NCBI Namespace Macros** The header *ncbistl.hpp* defines three principal namespace macros: `NCBI_NS_STD`, `NCBI_NS_NCBI` and `NCBI_USING_NAMESPACE_STD`. Respectively, these refer to the standard C++ `std::` namespace, a local NCBI namespace *ncbi::* for Toolkit entities, and a namespace combining the names from `NCBI_NS_STD` and `NCBI_NS_NCBI`.
- Using the NCBI Namespaces** Also in *ncbistl.hpp* are the macros `BEGIN_NCBI_SCOPE` and `END_NCBI_SCOPE`. These bracket code blocks which define names to be included in the NCBI namespace, and are invoked in nearly all of the Toolkit headers (see example). To use the NCBI namespace in a code block, place the `USING_NCBI_SCOPE` macro before the block references its first unqualified name. This macro also allows for unqualified use of the `std::` namespace. Much of the Toolkit source employs this macro (see example), although it is possible to define and work with other namespaces.
- Configuration-Dependent Macros and *ncbiconf.h*** `#ifdef` tests for the configuration-dependent macros, for example `_DEBUG` or `NCBI_OS_UNIX`, etc., are used throughout the Toolkit for conditional compilation and accommodate your environment's requirements. The **configure** script defines many of these macros; the resulting `#define`'s appear in the *ncbiconf.h* header and is found in the `<builddir>/inc` directory. It is not typically included explicitly by the programmer, however. Rather, it is included by other basic Toolkit headers (e.g., *ncbitype.h*, *ncbicfg.h*, *ncbistl.hpp*) to pick up configuration-specific features.
- NCBI Types (*ncbitype.h*, *ncbi\_limits.h* [*hpp*])** To promote code portability developers are strongly encouraged to use these standard C/C++ types whenever possible as they are ensured to have well-defined behavior throughout the Toolkit. Also see the current type-use rules. The *ncbitype.h* header provides a set of fixed-size integer types for special situations, while the *ncbi\_limits.h* [*hpp*] headers set numeric limits for the supported types.
- The *ncbistd.hpp* header** The NCBI C++ standard `#include`'s and `#define`'itions are found in *ncbistd.hpp*, which provides the interface to many of the basic Toolkit modules. The explicit NCBI headers included by *ncbistd.hpp* are: *ncbitype.h*, *ncbistl.hpp*, *ncbistr.hpp*, *ncbidbg.hpp*, *ncbiexpt.hpp* and *ncbi\_limits.h*.
- Portable Stream Handling** Programmers can ensure portable stream and buffer I/O operations by using the NCBI C++ Toolkit stream wrappers, `typedef`'s and `#define`'s declared in the *ncbistre.hpp*. For example, always use **CNcbiIstream** instead of *YourFavoriteNamespace::istream* and favor **NcbiCin** over **cin**. A variety of classes that perform case-conversion and other manipulations in conjunction with NCBI streams and buffers are also available. See the source for details.

- **Use of the C++ STL (Standard Template Library) in the Toolkit** The Toolkit employs the STL's set of template container classes, algorithms and iterators for managing collections of objects. Being standardized interfaces, coding with them provides portability. However, one drawback is the inability of STL containers to deal with reference objects, a problem area the Toolkit's **CRef** and **CObject** classes largely remedy.
- **Serializable Objects, the ASN.1 Data Types and datatool** The ASN.1 data model for biological data underlies all of the C and C++ Toolkit development at NCBI. The C++ Toolkit represents the ASN.1 data types as serializable objects, that is, objects able to save, restore, or transmit their state. This requires knowledge of an object's type and as such a **CTypeInfo** object is provided in each class to encapsulate type information. Additionally, object stream classes (**CObject[IO]Stream**, and subclasses) have been designed specifically to perform data object serialization. The nuts-and-bolts of doing this has been documented on the Processing Serial Data page, with additional information about the contents and parsing of ASN.1-derived objects in Traversing a Data Structure. Each of the serializable objects appears in its own subdirectory under `[src] include/objects`. These `objects/*` projects are configured differently from the rest of the Toolkit, in that header and source files are auto-generated from the ASN.1 specifications by the `datatool` program. The `--with-objects` flag to **configure** also directs a build of the *user classes* for the serializable objects.

## Key Classes

For reference, we list some of the fundamental classes used in developing applications with the Toolkit. Some of these classes are described elsewhere, but consult the library reference (Part 3 of this book) and the source browser for complete details.

- **CNcbiApplication** (abstract class used to define the basic functionality and behavior of an NCBI application; **this application class effectively supercedes the C-style main () function**)
- **CArgDescriptions**, **CArgs**, and **CArgValue** (command-line argument processing)
- **CNcbiEnvironment** (store, access, and modify environment variables)
- **CNcbiRegistry** (load, access, modify and store runtime information)
- **CNcbiDiag** (error handling for the Toolkit; )
- **CObject** (base class for objects requiring a reference count)
- **CRef** (a reference-counted smart pointer; particularly useful with STL and template classes)

- CObject[IO]Stream (serialized data streams)
- CTypeInfo and CObjectTypeInfo (Runtime Object Type Information; extensible to user-defined types)
- CObjectManager, etc. (classes for working with biological sequence data)
- CCgiApplication, etc. (classes to create CGI and Fast-CGI applications and handle CGI Diagnostics)
- CNCBINode, etc. (classes representing HTML tags and Web page content)
- Iterator Classes (easy traversal of collections and containers)
- Exception Handling (classes, macros and tracing for exceptions)

## The Object Manager and `datatool`

The `datatool` processes the ASN.1 specifications in the `src/objects/directories` and is the C++ Toolkit's analogue of the C Toolkit's `asntool`. The goal of `datatool` is to generate the class definitions corresponding to each ASN.1 defined data entity, including all required type information. As ASN.1 allows data to be selected from one of several types in a choice element, care must be taken to handle such cases.

The *Object Manager* is a C++ Toolkit library whose goal is to transparently download data from the GenBank database, investigate bio sequence data structure, retrieve sequence data, descriptions and annotations. In the library are classes such as **CDataLoader** and **CDataSource** which manage global and local accesses to data, **CSeqVector** and **CSeqMap** objects to find and manipulate sequence data, a number of specialized iterators to parse descriptions and annotations, among others. The **CObjectManager** and **CScope** classes provide the foundation of the library, managing data objects and coordinating their interactions.

An example, jump-start and *Object Manager* FAQ are all available to help new users.

## Debugging and Diagnostic Aids

The Toolkit has a number of methods for catching, reporting and handling coding bugs and exceptional conditions. During development, a debug mode exists to allow for assertions, traces and message posting. The standard C++ exception handling (which should be used as much as possible) has been extended by a pair of NCBI exception classes, **CErrnoException** and **CParseException** and additional associated macros. Diagnostics, including an `ERR_POST` macro available for routine error posting, have been built into the Toolkit infrastructure.

For more detailed and extensive reporting of an object's state (including the states of any contained objects), a special debug dump interface has been implemented. All objects derived from the **CObject** class, which is in turn derived from the abstract base class **CDebugDumpable**, automatically have this capability.

## Coding Standards and Guidelines

All C++ source in the Toolkit has a well-defined coding style which shall be used for new contributions and is highly encouraged for all user-developed code. Among these standards are

- variable naming conventions (for types, constants, class members, etc.)
- using namespaces and the NCBI name scope
- code indentation (4-space indentation, **no** tab symbols)
- declaring and defining classes and functions

## Noteworthy Files

A list of important files is given in Table 1.

**Table 1. Noteworthy Files**

Filename (relative to \$NCBI/c++)	Description
<b><i>configurecompilers/&lt;compiler_name&gt;.sh</i></b>	Use the <b><i>configure</i></b> shell script, or one of its compiler-specific wrappers, to fully configure and install all files required to build the Toolkit.
<b><i>scripts/cvs_core.{sh bat}</i></b>	Unix/Windows scripts to obtain the core, portable portions of the Toolkit from an existing CVS repository.
<b><i>scripts/import_project</i></b>	Import only an existing Toolkit project into an independent subtree of your current Toolkit source tree. (Requires a CVS source repository.)
<b><i>scripts/update_{core projects}</i></b>	Update your local copy of either the core Toolkit or set of specified projects. (Requires a CVS source repository.)
<b><i>scripts/new_project</i></b>	Set up a new project outside of the NCBI C++ Toolkit tree to access pre-built version of the Toolkit libraries. Sample code can be requested to serve as a template for the new module.
<i>src/&lt;project_dir&gt;/Makefile.in</i> <i>src/&lt;project_dir&gt;/ Makefile.</i> <i>&lt;project&gt;.{app, lib}</i>	Customized <i>meta-makefile</i> template and the corresponding datafile to provide project-specific source dependancies, libraries, compiler flags, etc. This information is accessed by <b><i>configure</i></b> to build a projects's <i>meta-makefile</i> (see below).
<i>doc/framewrk.{cpp hpp}</i>	Basic templates for source and header files that can be used when starting a new module. Includes common headers, the NCBI disclaimer and CVS keywords in a standard way.

Filename (relative to \$NCBI/c++)	Description
<i>CHECKOUT_STATUS</i>	This file summarizes the local source tree structure that was obtained when using one of the shell scripts in <i>scripts</i> . (Requires a CVS source repository.)
Build-specific Files (relative to \$NCBI/c++/ <builddir>)	Description
<i>Makefile Makefile.mk Makefile.meta</i>	These are the primary <i>makefiles</i> used to build the entire Toolkit (when used recursively). They are customized for a specific build from the corresponding *.in templates in \$NCBI/c++/src. <i>Makefile</i> is the master, top-level file, <i>Makefile.mk</i> sets many make and shell variables and <i>Makefile.meta</i> is where most of the make targets are defined.
<project_dir>/Makefile <project_dir>/ Makefile. <project>_{app, lib}	Project-specific custom <i>meta-makefile</i> and <i>makefiles</i> , respectively, configured from templates in the src/ hierarchy and any pertinent src/ <project_dir>/Makefile.<project>.{app, lib} files (see above).
inc/ncbiconf.h	Header that #define's many of the build-specific constants required by the Toolkit. This file is auto-generated by the <b>configure</b> script, and some pre-built versions do exist in <i>compilers</i> .
<b>reconfigure.sh</b>	Update the build tree due to changes in or the addition of configurable files (*.in files, such as <i>Makefile.in</i> or the <i>meta-makefiles</i> ) to the source tree.
<b>relocate.sh</b>	Adjust paths to this build tree and the relevant source tree.
corelib/ncbicfg.c	Define and manage the runtime path settings. This file is auto-generated by the <b>configure</b> script.
status/config.{cache log status}	These files provide information on <b>configure</b> 's construction of the build tree, and the cache of build settings to expedite future changes.